Programmeerimise põhikursus

ITI0010

Loengu sisu

Graafiline kasutajaliides: sissejuhatuse algus

Põhimõtteid

Paar näidet sissejuhatuseks

AWT (vana) ja Swing (uus) teegid

Iseseisev programm vs applet

Graphics and graphical user interfaces (GUI-s)

Drawing graphical shapes in your program is easy. You need to:

- Declare a place (example: a window) on screen where to draw
- Fill in corresponding pixels with right colours

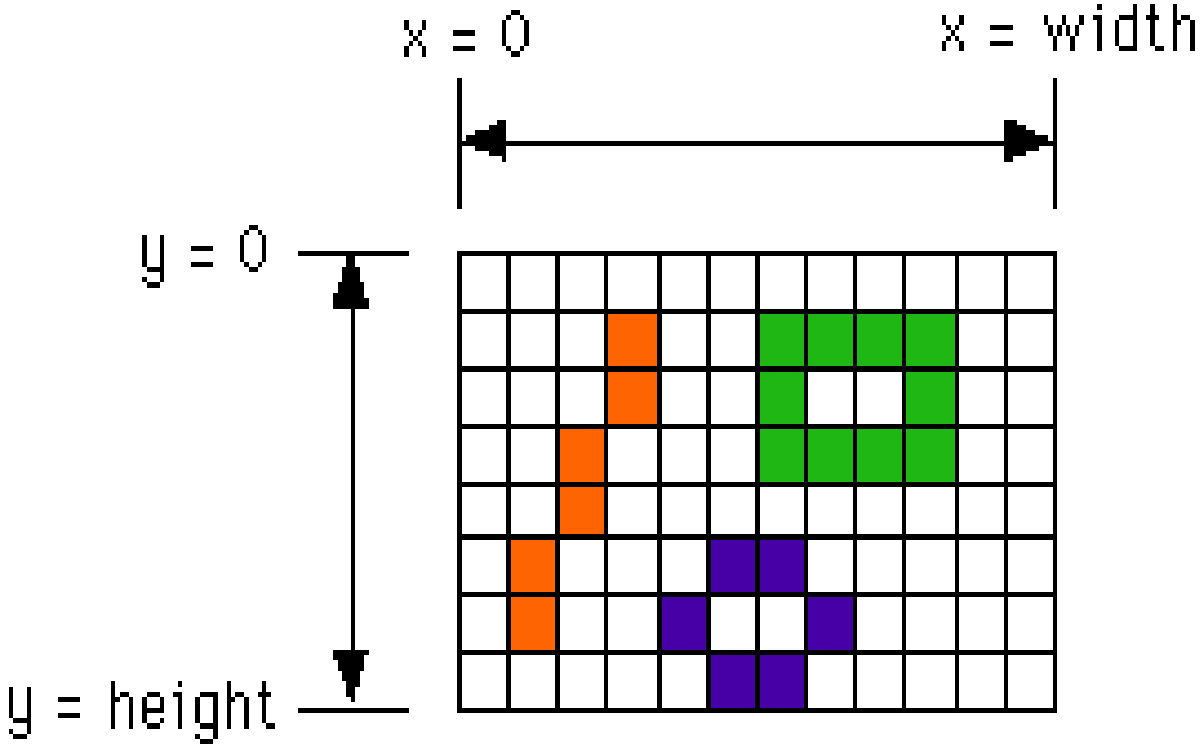
What is a bit harder is:

- Knowing how to fill in the pixels, actually
- Drawing complex shapes (circles, etc) fast
- Making graphics move fast (animation)
- Drawing all kinds of GUI widgets: buttons, menus, etc
- Understanding when has a user clicked on some region in the window (example, a button) and taking right action after that

Ready-made libraries help us to:

- Actually draw pixels
- Draw complex shapes
- Draw GUI widgets
- React to user mouse and keyboard actions on widgets
- Etc

Pixels and coordinates



Colors: RGB color system

An RGB color is specified by three numbers that give the level of red, green, and blue, respectively, in the color.

A color in Java is an object of the class, Color. You can construct a new color by specifying its red, blue, and green components. For example,

```
myColor = new Color(r,g,b);
```

There are two constructors that you can call in this way. In the one that I almost always use, r, g, and b are integers in the range 0 to 255. In the other, they are numbers of type float in the range 0.0F to 1.0F.

Often, you can avoid constructing new colors altogether, since the Color class defines several named constants representing common colors: **Color.white, Color.black, Color.red, Color.green, Color.blue, Color.cyan, Color.magenta, Color.yellow, Color.pink, Color.orange, Color.lightGray, Color.gray, and Color.darkGray.**

How to animate pictures?

A computer animation is really just a sequence of still images. The computer displays the images one after the other.

Each image differs a bit from the preceding image in the sequence. If the differences are not too big and if the sequence is displayed quickly enough, the eye is tricked into perceiving continuous motion.

We will look at animation programming later in the course.

Important things to understand about graphics

First, there are two kinds of ways to draw grahipcs:

Draw lines, circles, text, etc **where and how you will**, recognise user actions as you will.

Use pre-built graphical objects “**widget**s” which can also recognise user actions.

Second: the pre-built widgets are hierarchical classes. You can create their own subclasses and override the standard behaviour of a widget!

Third: on a window-based system all graphics is drawn on a window. However, the window may contain small subareas called “panels” or “containers”, which may contain their own small subareas, each independent. Thus, GUI is typically built of small independent blocks.

First example: the simplest window

```
import javax.swing.*;

public class HelloGraphics {

    // creates a small window with a text in it
    public static void main(String[] args) {
        // create a new window
        JFrame frame = new JFrame("HelloGraphics");

        // create a label widget with text, put on window
        JLabel label = new JLabel("Hello graphical world!");
        frame.getContentPane().add(label);

        // react to window close, pack stuff, show out
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();  // give a suitable size to window

                       //  automatically
        frame.setVisible(true);  // make window visible
    }
}
```

Second example: look-and-feel, panel

```
import javax.swing.*;
import java.awt.*;import java.awt.event.*;

public class HelloGraphics2 extends JPanel {
    static JFrame frame;
    // creates a small window with a text and a button in it
    public static void main(String[] args) {
        // create a new window and a new panel
        frame = new JFrame("HelloGraphics2");
        HelloGraphics2 panel = new HelloGraphics2();
        // add panel to the center of window
        frame.getContentPane().add("Center", panel);
        // Set a window look-and-feel
        try {
            UIManager.setLookAndFeel
              ("javax.swing.plaf.metal.MetalLookAndFeel");

         //"com.sun.java.swing.plaf.motif.MotifLookAndFeel");

        //("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
            SwingUtilities.updateComponentTreeUI(frame);
            frame.pack();
         }  catch (Exception exc) {
            System.err.println("Could not load!");
        }
```

Second example: look-and-feel, panel

```
        // react to window close, pack stuff, show out
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();  // give a suitable size to window
                       // automatically
        frame.setVisible(true);  // make window visible
    }

    // Class constructor: called when class is created

    public HelloGraphics2() {
        // create a label widget with text, put on panel
        JLabel label = new JLabel("Hello graphical world!");
        add(label);

        // Create and add a button to panel
        JButton button = new JButton("Press me, please!");
        add(button);
    }
}
```

Third example: let us look from editor

HelloGraphics3: does some real drawing as well
Typical code structure:

```
public class HelloGraphics3 extends JPanel {
  public static void main(String[] args) { // main: this starts
    // create a new window and a new panel, add panel to window
    frame = new JFrame("HelloGraphics3");
    HelloGraphics3 panel = new HelloGraphics3();
    frame.getContentPane().add("Center", panel);
    // determine window properties (size etc), show it
  }

  public HelloGraphics3() { // Class constructor
    // Create widgets and add to window
  }

  public void paintComponent(Graphics g) { // re-implement painting

    // For example, draw something on the Graphics type of object

    g.drawString("Hello is drawn now",100,200);
  }
}
```

From AWT to Swing

Java has many classes that work together to support a graphical user interface. These classes make up the Abstract Windowing Toolkit (**AWT**).

The classes can be found in the packages java.awt and java.awt.event.

The AWT is quite large and complex, more complex in fact than the basic Java programming language itself.

There is a new and greatly improved version of AWT, implemented in Java 1.2 as an addition to the existing AWT. It is called **JFC** or **Swing**. Swing widgets normally start with “J”, like this: **JApplet**, **JButton** etc

We will use JFC or Swing in this course!

NB! Several older browsers support old Java and do not support Swing.

Applications vs applets

There are basically two ways to run Java programs:

- **Applications**: as standalone applications started from command line
- **Applets**: as small programs running in web pages in a browser

We are mostly dealing with applications in this course.

However, the Eck book teaches GUI mostly on applet examples.

Fortunately, **applets are very similar to applications**!

However:

- Different browsers implement different, special versions of Java
- Many operations allowed for applications (reading/writing a file, opening an arbitrary network connection, etc) are prohibited for applets – for security reasons – while allowed for applications.
- **To summarise**: it is easier to create applications than fight the specific limitations and browser problems of applets.

Brief intro to applets: HTML basics

Applets generally appear in a Web browser program.

Such pages are themselves written in a language called HTML (HyperText Markup Language).

An HTML document describes the contents of a page. A Web browser interprets the HTML code to determine what to display on the page.

Tags:

The mark-up commands used by HTML are called tags. An HTML tag takes the form

<tag-name optional-modifiers>

Where the tag-name is a word that specifies the command, and the optional-modifiers, if present, are used to provide additional information for the command (much like parameters in subroutines). A modifier takes the form

modifier-name = value

Many tags require matching closing tags, which take the form

</tag-name>

Overall HTML document structure (may be simpler)

```
<HTML>
<HEAD>
<TITLE>page-title</TITLE>
</HEAD>
<BODY>
page-contents
</BODY>
</HTML>
```

The <appet> tag somewhere in page contents indicates that an applet has to be loaded and run. Example:

```
<applet code="HelloWorldApplet.class"
        width=200 height=50>
</applet>
```

Simple applet using AWT only

```
import java.awt.*;
import java.applet.*;

public class HelloWorldApplet extends Applet {
  // An applet that simply displays the string Hello World!

  public void paint(Graphics g) {
      g.drawString("Hello World!", 10, 30);
  }

}  // end of class HelloWorldApplet
```

Using mouse

Briefly look at SimpleStamper.java in editor

Main parts for mouse:

Class used MouseListener superclass in addition to Jpanel class:

```
Display extends JPanel implements MouseListener
```

MousePressed function is re-implemented

```
// The next four empty routines are required by the
// MouseListener interface.
public void mouseEntered(MouseEvent evt) { }
public void mouseExited(MouseEvent evt) { }
public void mouseClicked(MouseEvent evt) { }
public void mouseReleased(MouseEvent evt) { }
```

Converting an applet to an application

Let us briefly look at SimpleStamper1.java in editor

While the original class is an JApplet, it has been modified by TT to be a JFrame, that is, a standalone program.

The modification consists of:

- Changing the "... extends JApplet" to ".... extends JFrame"
- Adding the main function to the beginning of the class.
  - Create a frame
  - Call programmed init method on a frame
  - Determine frame properties, show frame
- Renaming SimpleStamper to SimpleStamper1, to avoid confusion.

Swing and AWT: components, layouts, and events

Components are the visible objects that make up a GUI. Every component is an object that belongs to some subclass of the class `java.awt.Component`. Some components are **containers**, which can hold other components. An applet is an example of a container.

Layouts. The components in a container must be "laid out," which means setting their sizes and positions. This is ordinarily done by a **layout manager**, which is an object associated with a container that implements some policy for laying out the components in that container.

Events are generated when the user interacts with a components. An event is represented by an object belonging to one of several classes in the `java.awt.event` package, such as `ActionEvent` and `MouseEvent`. An event has no effect unless a **listener** has been set up to listen for the event and respond to it.

Creating a user interface

Setting up a layout for the application.
Creating components and adding them to the application.
Arranging for listeners to listen for events.
Writing methods to respond when the events occur.

For each of these steps there are many options or cases to consider

Mouse events

In Java, events are represented by objects. When an event occurs, the system collects all the information relevant to the event and constructs an object to contain that information.

Different types of events are represented by objects belonging to different classes. For example, when the user presses a button on the mouse, an object belonging to a class called **MouseEvent** is constructed.

The object contains information such as the GUI component on which the user clicked, the (x,y) coordinates of the point in the component where the click occurred, and which button on the mouse was pressed.

Inherent **event loop**:

```
while the program is still running:
    Wait for the next event to occur
    Call a subroutine to handle the event
```

Mouse events

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SimpleStamper1 extends Jframe implements MouseListener {

  public void init() {
      // When the applet is created, set its background color
      // to black, and register the applet to listen to mouse
      // events on itself.
   setBackground(Color.black);
   addMouseListener(this);
  }
```

Mouse events

```
public void mousePressed(MouseEvent evt) {
  if ( evt.isShiftDown() ) {
    repaint();
    return;
  }
  int x = evt.getX();  // x-coordinate where user clicked.
  int y = evt.getY();  // y-coordinate where user clicked.
  Graphics g = getGraphics();  // get Graphics context
  if ( evt.isMetaDown() ) {
       // Draw a blue oval centered at the point (x,y).
     g.setColor(Color.blue);
     g.fillOval( x - 25, y - 15, 60, 30 );
     g.setColor(Color.black);
     g.drawOval( x - 25, y - 15, 60, 30 );
  }
  else {
       // Draw a red rectangle centered at the point (x,y).
     g.setColor(Color.red);
     g.fillRect( x - 25, y - 15, 60, 30 );
     g.setColor(Color.black);
     g.drawRect( x - 25, y - 15, 60, 30 );
  }
  g.dispose();  // We are finished with the graphics context,
 }  // end mousePressed()
```

Mouse events

```
  // The following empty routines are required by the
  // MouseListener interface:

  public void mouseEntered(MouseEvent evt) { }
  public void mouseExited(MouseEvent evt) { }
  public void mouseClicked(MouseEvent evt) { }
  public void mouseReleased(MouseEvent evt) { }


} // end class SimpleStamper
```

Converting an applet to an application

Let us look at SimpleStamper1.java in editor again, with all details

Painting on objects, Canvas

It is possible to draw directly on an applet or frame

However, it is not a good idea to do so when the frame/applet contains components that will be laid out by a layout manager. The reason is that it's hard to be sure exactly where the components will be placed by the layout manager and how big they will be.

A better idea is to add an extra component to the frame and do all the drawing on that component.

The `Canvas` class exists precisely for creating such drawing areas. An object that belongs to the `Canvas` class itself would be an empty "canvas." To create a canvas with content, you have to define a subclass of `Canvas` and write a `paint()` method for your subclass to draw the content you want.

When you use a canvas in this way, it's a good idea to put all the information necessary to do the drawing in the canvas object, rather than in the main object.

Canvas subclass

A new drawable object class:

The original colored hello world used an instance variable, `textColor`, to keep track of the color of the displayed message.

In the new version, the `textColor` variable is moved to the `ColoredHelloWorldCanvas` class.

This class also contains a method, `setTextColor()`, that the application can use to tell the canvas to change the color of the message.

Good object-oriented program design:

The `ColoredHelloWorldCanvas` class is responsible for displaying a colored greeting, so it should contain all the data and behaviors associated with its role.

This class doesn't need to know anything about buttons, layouts, and events.

Canvas subclass

```
class ColoredHelloWorldCanvas extends Canvas {
  Color textColor;  // Color in which "Hello
                    // World" is displayed.
  Font textFont;   // The font in which the
                    // message is displayed.

  ColoredHelloWorldCanvas() {
     // Constructor.
      setBackground(Color.white);
      textColor = Color.red;
      textFont = new
                 Font("Serif",Font.BOLD,24);
   }

   public void paint(Graphics g) {
      // Show the message
      g.setColor(textColor);
      g.setFont(textFont);
      g.drawString("Hello World!", 20,40);
   }

   void setTextColor(Color color) {
     // Set the text color and tell the system
     // to repaint the canvas.
      textColor = color;
      repaint();
   }
}  // end class
```

Buttons and other widgets which do something

Look at HelloWorldJApplet in editor

public class HelloWorldJApplet extends JApplet implements ActionListener

.....

One of the techniques in the source:

Each button is associated with a function

When button is pressed, the function is automatically called

The other important technique:

When something is added to the container, we will explicitly say HOW: by using layouts

A more complex init

```
public void init() {
  // It creates the canvas and lays out the
  // applet to consist of a
  // bar of control buttons below the canvas.
  setBackground(Color.lightGray);
  canvas = new ColoredHelloWorldCanvas();
  Panel buttonBar = new Panel();  // a panel
             // to hold the control buttons
  // Create buttons and add them
  Button redBttn = new Button("Red");
  // button bar.
  buttonBar.add(redBttn);
  Button greenBttn = new Button("Green");
  greenBttn.addActionListener(this);
  buttonBar.add(greenBttn);
  Button blueBttn = new Button("Blue");
  blueBttn.addActionListener(this);
  buttonBar.add(blueBttn);
  // Lay out the applet
  setLayout(new BorderLayout(3,3));
  add("Center",canvas);
  add("South", buttonBar);
}  // end init()
```

inside init: creating buttons

The line,

```
Button redButtn = new Button("Red");"
```

creates a button labeled "Red."

Once the button has been added to the applet, it will pretty much take care of itself.

The statement "`redBttn.addActionListener(this)`" tells the button that the applet (referred to by this) wants to be informed whenever the user clicks on the button.

When the user clicks on the button, the button will call the applet's `actionPerformed()` method to let the applet know about it.

We say that the applet is listening for action events from the button. Any object can be an action listener for a button, provided it implements the `ActionListener` interface and defines the `actionPerformed` method.

The last thing we do with `redBttn` is to add it to the `buttonBar`. The "Green" and "Blue" buttons are handled the same way.

inside init: BorderLayout

Now that the canvas and button bar have been created, it's time to lay out the applet as a whole.

This is done by the last three lines of the `init()` method.

We use a "BorderLayout," which can display one big component in the "Center" of the applet and up to four other components along the edges of the applet to the "North", "South", "East", and "West".

In this case, the canvas is added in the "Center" position, with the button bar below it, to the "South".

a new actionPerformed method

```
public void actionPerformed(ActionEvent evt) {
   String command = evt.getActionCommand();

   if (command.equals("Red"))
      canvas.setTextColor(Color.red);
   else if (command.equals("Green"))
      canvas.setTextColor(Color.green);
   else if (command.equals("Blue"))
      canvas.setTextColor(Color.blue);

}
```

inside actionPerformed

The parameter, evt, of this method is an object that carries information about the particular event that caused the routine to be called.

Since the same method is called no matter which of the three buttons was clicked, we need to find out which button it was.

The method evt.getActionCommand() returns the name of the button that was clicked (unless you've configured the button to send a different string).

In this example, the command must be either "Red" or "Blue" or "Green".

Layouts

See the code for LayoutDemo.java

More components

Let us see the code for GUIDemo2.java

Mouse movements

Let us see the code for SimplePaint2.java

paint, repaint, and ...

Many components do all their drawing operations in their paint() methods. The **paint()** method should be smart enough to correctly redraw the component at any time, using data stored in instance variables that record the state of the component.

If, in the middle of some other method, you realize that the appearance of the component should change, you should change the values of those instance variables and call the component's **repaint()** method, which tells the system that it should redraw the component as soon as it gets a chance (by calling the component's paint() method).

update

A complication arises from the fact that system does not actually call the paint() method of a component directly.

There is another method called update() which is the one actually called by the system. The built-in update procedure first fills in the entire component with its background color. Then it calls the paint() method to redraw the contents.

Usually, erasing the component first is the right thing to do, since the contents of the component might have changed. However, in some cases you will want to avoid this step.

In that case, you can override update() to read simply:

```
public void update(Graphics g) {
   paint(g); // call paint, without erasing
}
```